Unwanted Behavior and its Impact on Adaptive Systems in Ubiquitous Computing

Michael Fahrmair, Wassiou Sitou, Bernd Spanfelner
Technische Universität München – Department of Informatics
Boltzmannstr. 3, 85748 Garching (Munich), Germany
{fahrmair|sitou|spanfeln}@in.tum.de

Abstract

Many ubiquitous computing applications so far fail to live up to their expectations. While working perfectly in controllable laboratory environments, they seem to be particularly prone to problems related to a discrepancy between user expectation and systems behavior when released into the wild. This kind of unwanted behavior of course prevents the vision of an emerging trend of context aware and adaptive applications in mobile and ubiquitous computing to become reality.

In this paper, we present examples from our practical work and show why for ubiquitous computing unwanted behavior is not just a matter of enough requirements engineering and good or bad technical system verification. We furthermore provide a classification of the phenomenon and an analysis of the causes of its occurrence and resolvability in context aware and adaptive systems.

1 Introduction

The main intention of ubiquitous computing (Ubicomp) is the use of functionality in as many situations as possible [Fahrmair, 2005; Dey, 2000; Weiser, 1991; Schmidt, 2002]. Context adaptation in this setting is an enabling technology for ubiquitous computing since it allows a technical system to change its structure, functionality or implementation at runtime to adapt to situation depending conditions. *Context* in this scope means the sufficiently exact characterization of a system's situation by means of perceivable information that is relevant for the adaptation of the system (a model of a situation). *Adaptation* again is a term to describe the general ability to fit to different conditions or circumstances given by the environment in a certain situation. *Context adaptation* is therefore shortly defined as an automated adjustment of the observable behavior or the internal states of a system to its context [Fahrmair *et al.*, 2006].

While classic non ubiquitous applications are used like hammers and screwdrivers and let the user decide how to put up a picture, ubiquitous applications are comparable to discreet servants. Our experiments clearly showed however that this paradigm shift seems to intensify problems related to user acceptance with such systems and their applications. Identifying and tracking back the causes, we observed that besides technical errors there was a new class of failure type involved. Sometimes the system even behaved exactly as intended by the developer, however for a certain combination of user, his situation and the environment, this behavior could be seen as wrong and even disadvantageous despite other users might have been perfectly satisfied.

To better distinguish this effect from failures that are undisputable an error, no matter what perspective or special situation you look at it from, we introduced the term *unwanted behavior* (UB). The main difference of this unwanted behavior to, for example, specification/implementation errors is that unwanted behavior can not be systematically detected by means of typical verification techniques like theorem proofing or model checking. In other words it is possible to have a 100% correct system that is still completely useless for a specific user in a specific situation. However since the system can work perfectly for most of all users most of the time, it is also not just a question of good or bad requirements engineering. Moreover since this phenomenon especially happens when releasing an application from the safety of the development lab (with its controllable environment) out into an infinite complex and unpredictable real world, this is also not a matter of just finding the right test cases during the development phase.

Context awareness is fine in theory, and the research issue is figuring out how to get it to work. Therefore, two applications, Grapevine and Rendezvous, developed and deployed by IBM, have revealed the key challenges in making context-aware computing a reality [Christensen *et al.*, 2006]. Grapevine and Rendezvous are services offered to IBM employees as a means of looking into the promise and perils of context-aware computing. The majority of users, however, did not find application activity context useful for a variety of reasons. For example, users often were not comfortable with others knowing what they were doing. The Grapevine [Richards and Christensen, 2004] service provided complete control over who could observe which elements of context, and users commonly blocked all others from viewing their computer activity all of the time. Although the service allowed observer-by-observer blocking, it was rarely used. The IBM Rendezvous service allows people to talk in small groups using telephones (to "rendezvous" on the phone) and/or computer applications that provide a telephone function. This is similar to audio-conference calls, and the service appears to be a layer on top of audio-conferencing. Instead of calling directly into an audio-conference, however, a user of the IBM Rendezvous service in effect phones his or her corporate calendar, selects a meeting from it, and enters into a multi party conversation with the people invited to that meeting.

In this paper we put all these and other observations and assumptions on a firmer theoretical and practical basis and discuss their technical and methodical resolvability for the domain of software and systems engineering:

- We give a brief overview (Section 2) why unwanted behavior is an intensified problem for ubiquitous applications that are realized with context adaptation. We describe both examples of existing real life as well as academic prototypes and concepts to illustrate the above mentioned unwanted behavior problems (Section 3)
- Although it is superficially possible to classify the described examples into three different types of failure reasons, our analysis (Section 4) shows that the real cause for UB is always a divergence between two models of reality.
- There are exactly three sources for such model divergences. We explain why such discrepancy can survive most technical quality measures, stay hidden in the system for a long time or even spontaneously arise while running the system (Section 5). We also do a short discussion on what constructive or methodical measure can be taken to avoid unwanted behavior and how these strategies worked out in our practical applications.

2 Impact of Unwanted Behavior (UB) on Ubiquitous Computing

The main idea behind all these concepts in particular and of ubiquitous computing in general is a more flexible system understanding, whereby the thought of the system as a tool moves into the background and the needs and wishes of the user step into the foreground. Generally, these needs and wishes of a given user vary according to his current situation.

During the last six years we have developed and experimented with several prototypes of applications based on that idea of systems that can automatically recognize wishes and needs of its users (or other stakeholders) and adapt themselves accordingly by means of reconfiguration. Among them were a mobile community based search engine [Fahrmair *et al.*, 2000], an in-house navigation and information assistant for a campus [Amann *et al.*, 2004] and a one-year long self experiment with a smart home environment [Fahrmair, 2005].

From the introducing description of UB, there seems to be no reason why UB cannot arise in non ubiquitous systems. In fact this assumption is true. However ubiquitous applications usually

C1: are multi functional and more complex, and operate in heterogeneous environments,

C2: work, at least to some extent, invisibly in the background, and

C3: are technically based on automation (context adaptation)

Because of C1, Ubicomp systems are much less transparent for their users, especially regarding technical limitations of possible functions. Due to C2, there is no such thing like an operating error in Ubicomp. Therefore the user rightly insists that a Ubicomp application fulfills his wishes and needs as promised and not just does what he has explicitly commanded. This difficulty even gets worse if the system relies on automation (C3) to fulfill its ubiquity goal, even in situations with very limited user interaction possibilities (e.g. while driving a car).

This is because, even if UB can also occur in normal systems, they usually have a spontaneous hull [Raasch, 1993] that can help to prevent or compensate for negative UB experience. A spontaneous hull in short is some kind of interactive influence sphere around the actual technical system core. This user influence can be used to compensate for known or anticipated UB for example by manually modifying input and output values, finding operational workarounds for bugs etc. With increasing automation and decreasing user interaction resources in ubiquitous systems, this mechanism however can no longer compensate UB below a tolerable level.

3 Examples of Unwanted Behavior

In this section we provide a couple of examples illustrating the occurrence of unwanted behavior while dealing with software systems, particularly if the systems possess certain automatic (i.e. context adaptive behavior). Some of these examples are constructed. Since not many ubiquitous systems have been released, the examples aim to illustrate unwanted behavior also in common applications. We should point out that in Ubicomp the occurrence of UB is however amplified by the ubiquitousness of the applications.

3.1 Microsoft Office Spell Checking

A well-known, yet very simple, example of unwanted behavior is the automatic spell checker incorporated in MS Office Word. After a full stop, the first letter of the following word is automatically capitalized. In fact, this function cannot differentiate between abbreviations with point and the real ending of a sentence. If this fact is not familiar to a given user, the written text could hold some surprises during a read-after-write check. This is an example for situations, where the system is not able to predict the users intentions. Of course no one would care about this if it occurs once in a while but, as Ubicomp applications are meant to support the user in as many situations as possible with automated decisions, such small glitches can sum up to a big annoyance.

3.2 Smart Kitchen

A nearly inexhaustible source for effects of unwanted behavior are smart home applications as described in [Fahrmair, 2005]. This occurs particularly if the user can not understand the exact technical realization of the system. A typical example concerns smart kitchens with their underlined automatic food order systems. A lowbrow user for instance could develop the impression that the ordered food is only based on the editable purchasing list. In fact this observation is however only a coincidence, which arises as a result of the fact that the user does not transact any additional purchases, which would be registered by the system over the RFID labels and considered for new orders. If the user suddenly changes his relevant behavior, for example by adjustment of a weekly poker party, for which the guests bring some food and put it in the refrigerator, this dependence hidden so far can lead to terrible surprises. The system would register the regular consumption of additional goods and would adapt the automatic order accordingly. This example highlights problems that occur because of a misunderstanding of the systems inner behavior. More accurate or better understandable documentation in this case is not necessary sufficient, because Ubicomp systems adapt their structure and functionality at runtime.

3.3 Navigation System

Lets consider a common GPS-based navigation system. Such a system has as primary task to guide its user from a location A to another location B. At the beginning of the guidance, the system computes the route. Thereby it considers in addition to the current input of the user, also his preferences, and guides him to the desired destination. If the user gets lost during the guidance, the system recomputes the route from the current position to the destination. In our example, a building site that recently began is on the computed route. This building site however is not yet taken up into the street guide. The system is therefore unable to recognize this new situation and guides the user into this dead-end street. The user is unsatisfied, turns 400m back and takes another way. After a while, the user lets the navigation system guide him again to his destination with the hope the detour would cause the system to choose another route. Once again, the system guides the user to the temporary dead-end street at the building site. The user had to go back even further to let the system definitively avoid the building site. Of course there are already new navigation systems that allow the user to exclude specific streets. This requires additional user interactions in a situation where the user is already busy driving the car. This example illustrates unwanted behavior, where the system, in contrast to the user, does not have the necessary abilities to detect exceptional situation.

3.4 Further Examples: Air-conditioning, Bank Service

Other examples for unwanted System behavior would be a climate control that does adapt to the cultural background and a user that has guests from abroad and suddenly is disappointed about the system adapting to the guests, due to the majority criterion of the adaptation logic, and not to him. Here the user was used to a certain behavior that apparently changes spontaneously. A further example is an automatic savings function for the bank account that saves money above a certain threshold and accidentally saves money that someone has assigned on the account to pay a certain anticipated bill.

4 Characterization of Unwanted Behavior

We use the term *unwanted behavior* to designate the phenomenon where the behavior of a given system, while free of errors, still differs from the expectations of its current user.

4.1 Criteria for the Occurrence of UB

Analyzing the above mentioned motivation and examples, we summarize that unwanted behavior occurs if the following criteria are all together fulfilled:

- There exists an observable divergence between user expectation and system behavior.
- The system behaves correctly regarding its specification
- The system specification complies with the collected requirements.

The existence of the observable divergence between user expectation and system behavior is with reference to the above mentioned observations undisputable. The idea of drawing a distinction between user model and systems model is similar to Norman's canonical elucidation of the role of mental models in the design process [Norman, 1988]. Norman states that the designer's goal is to design the system image such that the user's mental model of the system's operation coincides with the designer's mental model of the same. The system image represents those aspect of the implementation with witch the user interacts. Yet the above mentioned discrepancy, i.e the observable divergence between user expectation and system behavior, could not be seen as system construction failure, since the system exactly behaves as specified by the engineers. The main cause of the occurrence of such unwanted behaviors therefore seems to be a lack in collecting and processing the users needs and wishes.

We derive from these criteria that an unwanted behavior occurs if, on the one hand, the user is not aware of the system's abilities and thus develops expectations that are unrealizable by the system. On the other hand, UB occurs if the needs of the user are altered due to external influences, which are unrecognizable by the system. Over and above that, UB could be registered if situations (or context) of use arise at runtime, which were not predictable at development time. The system then proceeds from assumptions that might not be valid any longer and thus behaves suddenly incorrectly from the users point of view. In this way UB-occurrences are events that individually usually are not critical for the overall functionality of a system. In larger quantities, UB can become a growing annoyance though. This can lead to rejection by the user and therefore to a replacement of a system.

4.2 Cause of Occurrence of UB

Events that meet our definition of UB can be traced back to three reasons (see Figure 1).

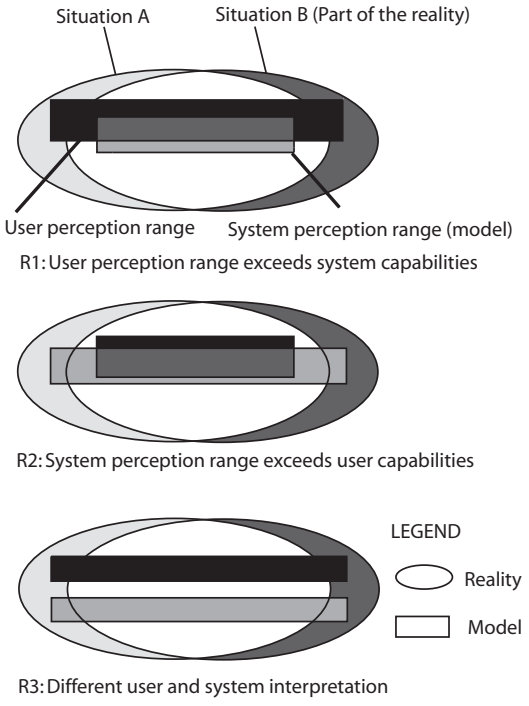

Figure 1: Differences in the Perception of Different Situations

These reasons are sometimes sole responsible for a UB-Phenomenon but can also occur in combination:

R1: The user is able to differentiate two situations, but the system is not.

R2: The system is able to differentiate two situations, but the user is not.

R3: Both, user and system are able to differentiate the situations but they have different interpretation or perspective of the situation's changes.

R1 is typical for ubiquitous systems that lack needed sensors to identify a certain situation. This can either be the case if the situation was not considered during the requirements engineering and hence a required sensor was not integrated in the system. Another possibility is that sufficient exact sensors are not available, or even if so, they are too expensive etc. In either case the context information that is available to the system is not sufficiently exact to characterize all relevant situations. Since context is an abstraction of the reality where irrelevant details are dropped, a third possibility for R1 could be that the system model is too abstract and details that are important for the user may have been dropped.

R2 is the exact opposite of R1. If a system uses sensors that observe the environment in terms that are beyond the perception of the user, it is possible that the system will identify a new situation where the user is not aware of any changes. Another possibility could be that the system model is too detailed. Sometimes little changes in the environment are perceptible by a user per se, but in normal life these details are filtered out. If the system model observes such details (for example in discrete rules) it is likely that the system identifies a change of the situation but the user does not. Also the system designer could be of the opinion that a certain change in the environment leads to a different situation but the user does not share this opinion. Since ubiquitous systems are commodity, it is likely that some user's opinion about different situations differ from the designer's opinion. In contrast to a too detailed model where the designer did not bear in mind that certain details are irrelevant to a user, here there is a basic difference in identifying situations.

R3 is a bit related to the different interpretation of a situation. Here the user as well as the system designer had different interpretations of a situation. The main difference compared to the last reason is that in the former argument the difference lies in the importance to distinguish between two situations whereas now both agree that a new situation is recognizable but they differ in the interpretation of the new situation. This could be the case if, for instance, the cultural backgrounds of the designer and the user are different.

Despite the reason for UB phenomenon differs, there is a common fact that is equal to all three reasons: The user's reality model, from which he derives his wishes and needs, differs from the system model that represents the reality model of the designer.

5 Model Discrepancies (MD) and their Origins

Now that we have identified MDs as the main reason of UB in an otherwise technical error free system, the interesting question is how and when such MDs are created and whether they can be detected, removed or at least avoided or compensated for.

To understand the difficulty of this question we have to make clear that this question is not about comparing two technical models, which is often done in software verification (e.g. checking design specification model against implementation) and is at least theoretically feasible. The UB problem is about comparing models with reality, i.e. checking whether the assumptions that were used to *create* the model in an abstraction process have been valid in the first place and are still valid. In ubiquitous computing this means comparing a black box model of yet unknown type and structure inside the users head with a similar model of the developer of the system. While the model of the developer is at least partially visible in form of the system implementation, the user model is not. To make it more difficult, the developer's model representation is frozen at a certain time during the development phase while the user's world model constantly changes and should be updated to reflect any unforeseeable changes in reality, his growing experience etc. Yet the user's model is still only a projection of reality from a certain perspective and of course this perspective is not necessarily the same as the developer's.

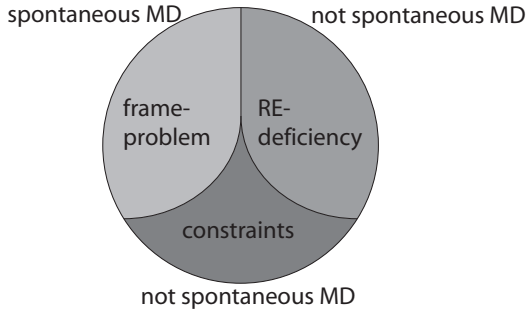

Figure 2: Three Reasons for Model Discrepancies

It follows from this that MDs either already exist at development time (hidden MDs) or (spontaneously) emerge during the system lifetime (see Figure 2). The latter possibility is not as obvious so we first take a look at hidden MDs that are introduced somewhere within the development process. Our analysis of several case studies clearly showed that these hidden MDs can have two possible origins.

The first possibility is that the UB is caused by a MD that originated from a *conscious decision by the system designer*. For example, the designer realized that among 1000 possible usage contexts there might be 2 with use cases and requirements conflicts that would need an uneconomical amount of effort to be resolved. So the decision was made to deliberately drop support for this situation. However the first problem with such decisions in a ubiquitous application is that ubiquitous systems usually are extended, recombined or depended on in an unforeseeable way. Even if a failure rate of 0.2% seems to be pristinely tolerable, such effects can propagate and multiply due to service composition (typical for Ubicomp systems) rendering later applications of the system pretty useless. It could be argued that this problem is more or less a matter of writing and reading proper manuals. Yet the problem is that even if this decision was explicitly made, even documented, this information does not always make it through the development process. Even if it would in Ubicomp (because of its dynamic nature) there is usually no printed manual.

The second possibility for hidden MDs is *deficits in requirements engineering* (RE). In our terminology we deliberately speak of deficits instead of errors because we distin-

guish between real errors (like registering and documenting a requirement in a document that is later on overwritten by another version or skipping that annoying interview etc.) and "did not know it better at that time" effects. So the reason is that state of the art RE methods seem to be not fully suitable for the construction of complex Ubicomp applications that work in highly heterogeneous dynamic environments.

As mentioned before there is a third possible reason for MD. This one is especially though to handle because the MD does not exist during development time but arises later. This last condition however is not undisputable. The whole matter is under heavy discussion for more than 20 years especially in the field of AI (known there as the frame problem [Dennett, 1984]). However there seem to be a lot of good arguments that this problem is not generally, and now less then ever practically, solvable at least until it is possible to give a machine at least limited abilities in mind reading and fortune telling. Another possibility would be to avoid using classical model representations (with extrinsic semantic) at all. Using a model representation based on self contained semantic (like the mathematic language is for mathematics) to reflect a significant part of the real world however would most likely mean constructing a machine with higher complexity than our universe.

We therefore assume that there are certain random events within reality or the user's thoughts and ideas that can cause MDs at a later point in time than the development of the system (see Figure 3). Moreover because of the symbol grounding problem [Harnad, 1990] such MDs can not be detected from inside the model without any outside help. This means MDs can stay hidden until they cause UB or at least become imminent.

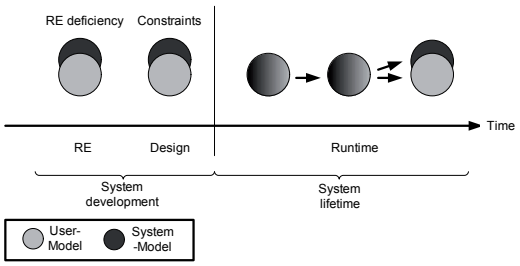

Figure 3: Model Discrepancies and their Origins

6 Related Terms

While dealing with the phenomenon described above, the question arose of whether we really need a new identifier? And is there any relation to terms like classical system error, operating error, feature interaction, automation surprise or software aging?

6.1 Classical System Error

An often asked question regarding the whole issue of unwanted behavior caused by model divergences is, whether this is not just some kind of classic error. Especially because for instance, an implementation error could also lead to wrong results in a software system, which is of course also disliked by the user.

The main difference however is that such errors usually are unacceptable for all users no matter what situation they are in. Also a system with errors is usually regarded as incorrect compared to a formal specification, which makes errors at least theoretically possible to dispel at development time by various means of different verification techniques.

Our definition of unwanted behavior therefore purposely excluded such technical errors because we differentiate between failures that can be prevented by means of verification and failures that can not be systematically detected during development time. Basically this is inspired by the fact that there can be systems that are 100% correct but can still be completely useless in a certain situation. This is also why MDs can not always be detected at development time. Unlike classical system errors, MDs are hidden in implicit assumptions (in other words the abstractions) that were used to create a model in the first place.

6.2 Operating Error

The only other form of errors not covered by this differentiation between UB and specification errors are operating errors. These describe failure situations where the system would depend on correct user interaction to produce a correct output behavior. Ubiquitous applications however usually work invisibly in the background, at least to some extent and therefore have to be foolproof by definition, at least regarding any remaining direct user input. This is why we neglect operating errors as a source of UB provided that they are not already covered by MDs in general, for example when the user has developed a wrong idea about the capabilities of the system.

6.3 Feature Interaction and Automation Surprises

Besides classical system and operating errors there is yet another term that is often confused with MD created UB. *Feature interaction* however relates to a large group of mostly technical issues around interaction and combination of features (for a good overview see [Pulvermüller *et al.*, 2002]). Inside this group especially unwanted feature interactions not caused by specification/implementation errors come quite close to unwanted behavior, but MDs are not limited to a combination or interaction of features. Therefore feature interaction problems that are not related to errors are a subset of our definition of UB.

The same is true for *automation surprises* or so called *mode errors* [Hourizi and Johnson, 2001]. These are also small subsets of UB that especially concentrate on user interfacing aspects. Moreover automation surprises only concentrate, on system behavior non-transparency and makes no difference between unexpected and unwanted behavior. Mode errors again require at least some sort of direct and conscious user interaction, which is not always the case for ubiquitous systems. We therefore regard both terms as very special cases of UB. In fact they describe rare cases of non ubiquitous applications where UB can have a larger impact (e.g. in avionic systems).

6.4 Software Aging

Finally the term *software aging*, while seeming quite related especially to MDs accrued during the system's lifetime, usually focuses on increasing difficulties to change an old system to new requirements (or to cope with an increasing number of such changes afterwards [Parnas, 1994]). The reasons that requirements can become obsolete of course are the same as MDs appearing during runtime of an ubiquitous system. The main difference between MD based UB and software aging however is that the latter

needs several years or even decades of lifetime to evolve relatively stable changes in the wishes and needs of their users while spontaneous MDs in ubiquitous applications can happen during runtime and can be instable. Also they usually heavily depend on the specific user and his situation, while software aging usually refers to a collective change in requirements (e.g. like supporting a new technology, platform or process).

7 Conclusion

In this paper, the phenomenon of unwanted behavior (UB) has been characterized and examples from our practical work have been given. We have also provided a classification of the phenomenon and analyzed the causes of its occurrence and resolvability, particularly in the filed of context awareness and ubiquitous computing.

While researching user acceptance problems in real life ubiquitous computing applications, there always has been one question at issue: Is the problem of unwanted behavior really a new kind of problem? Is it a specific drawback of ubiquitousness or a question of good (or bad) requirements engineering? Unfortunately the answer is more complicated than the question. Can good-enough requirements engineering prevent from UB? Yes and no. Yes, because the main goal of RE is to analyze the user wishes and needs and prepare them to be translated into a technical specification for system design. No, because wishes and needs change over time depending on necessities of reality. Therefore, while analyzing requirements, at least enough resources, a good clairvoyance and adequate methods would be needed. But, what is exactly "good enough" RE? What is it all about? For ubiquitous computing this means the more wide the application domain gets (both in lifetime and complexity), the more probable it is that there exist discrepancies between user expectation and system behaviors. Each time this happens, a UB will be created. This is a user experience problem in Ubicomp because of the claim to fulfill user wishes and needs without necessary direct or conscious interaction. Unconscious usage means automation and hence less possibility to mediate or real-time correction by the user.

However the system model could have been wrong from the beginning due to deficits in RE, but also MDs can arise later due to framing problems. Because the user is also usually a customer of some sort, for him it is not his concern if this UB is caused by a bad RE analyst or a tricky twist in reality. Therefore it is necessary to at least introduce another compensation mechanism to replace any real-time compensation possibilities the user had in non-automatically acting applications. This mechanism is called calibration and described in [Fahrmair, 2005; Fahrmair *et al.*, 2006; Newberger and Dey, 2003]. Once again, this mechanism should be no excuse for relaxed RE. There is a number of reasons why as much effort should be spent on RE as economically possible to reduce the number of UB caused by failed RE since the calibration is not a totally preventive mechanism (every engineer probably used to sit in an airplane from time to time). However there are also remarkable indications for not delivering ubiquitous application without calibration support. And last but not least there are also a number of good reasons to not scrap ubiquitous computing all together because for instance it allows computers to extend their full potential from direct human machine interactions to sub- or semi-conscious secondary usage in almost every situation.

References

[Amann *et al.*, 2004] K. Amann, T. Reichgruber, and M. Roming. *Personalisierung kontextadaptiver Dienste*. Student Project, Technische Universität München, 2004.

[Christensen *et al.*, 2006] J. Christensen, J. Sussman, S. Levy, W. E. Bennett, T. V. Wolf, W. A. Kellogg. *Too Much Information*. HCI, ACM Queue 4(6), 2006.

[Dennett, 1984] D. C. Dennett. *Cognitive Wheels: The Frame Problem of AI* . Ed.:C. Hookway, Minds, Machines, and Evolution. Cambridge University Press, Cambridge, 1984.

[Dey, 2000] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD Thesis, College of Computing, Georgia Institute of Technology, 2000.

[Fahrmair, 2005] M. Fahrmair. *Kalibrierbare Kontextadaption für Ubiquitous Computing.* Dissertation, Department of Informatics, Technische Universität München, 2005.

[Fahrmair *et al.*, 2000] M. Fahrmair, C. Salzmann, and M. Schoenmakers. *Verfahren zur Vorauswahl mobiler Dienste*. IPR DE0010024368A1 [DE], DPMA, 2000.

[Fahrmair *et al.*, 2006] M. Fahrmair, W. Sitou, and B. Spanfelner. *An Engineering Approach to Adaptation and Calibration*. Modeling and Retrieval of Context MRC 2005, Ed.: T. Roth-Berghofer, S. Schulz and D. Leake, LNCS 3946, 2006.

[Harnad, 1990] S. Harnad. *The Symbol Grounding Problem*. Physica D 42, 1990.

[Hourizi and Johnson, 2001] R. Hourizi, and P. Johnson. *Beyond Mode Error: Supporting Strategic Knowledge Structures to Enhance Cockpit Safety*. Joint Proc. HCI 2001 and ICM 2001.

[Newberger and Dey, 2003] A. Newberger and A. K. Dey. *Designer Support for Context Monitoring and Control*. IRB-TR-03-017, Intel Research Berkeley, 2003.

[Norman, 1988] D. A. Norman *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[Parnas, 1994] D. L. Parnas. *Software Aging*. 16th Int. Conf. on Software Engineering (ICSE-16), 1994.

[Pulvermüller *et al.*, 2002] E. Pulvermüller, A. Speck, J. O. Coplien, M. D'Hondt, and W. DeMeuter. *Feature Interaction in Composed Systems*. LNCS 2323, 2002.

[Raasch, 1993] J. Raasch. *Systementwicklung mit Strukturierten Methoden. Ein Leitfaden für Praxis und Studium*. Hanser, 3. Auflage, München, Wien, 1993.

[Richards and Christensen, 2004] J. Richards and J. Christensen. *People in our Software*. ACM Queue 1(10), 2004.

[Schmidt, 2002] A. Schmidt. *Ubiquitous Computing - Computing in Context*. PhD Thesis, Computing Department, Lancester University, U.K., 2002.

[Weiser, 1991] M. Weiser. *The Computer for the 21st Century*. Scientific American, 1991.